What's Really New with NewSQL?

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Matthew Aslett
451 Research
matthew.aslett@451research.com

ABSTRACT

A new class of database management systems (DBMSs) called **NewSQL** tout their ability to scale modern on-line transaction processing (OLTP) workloads in a way that is not possible with legacy systems. The term NewSQL was first used by one of the authors of this article in a 2011 business analysis report discussing the rise of new database systems as challengers to these established vendors (Oracle, IBM, Microsoft). The other author was working on what became one of the first examples of a NewSQL DBMS. Since then several companies and research projects have used this term (rightly and wrongly) to describe their systems.

Given that relational DBMSs have been around for over four decades, it is justifiable to ask whether the claim of NewSQL's superiority is actually true or whether it is simply marketing. If they are indeed able to get better performance, then the next question is whether there is anything scientifically new about them that enables them to achieve these gains or is it just that hardware has advanced so much that now the bottlenecks from earlier years are no longer a problem.

To do this, we first discuss the history of databases to understand how NewSQL systems came about. We then provide a detailed explanation of what the term NewSQL means and the different categories of systems that fall under this definition.

1. A BRIEF HISTORY OF DBMSS

The first DBMSs came on-line in the mid 1960s. One of the first was IBM's IMS that was built to keep track of the supplies and parts inventory for the Saturn V and Apollo space exploration projects. It helped introduce the idea that an application's code should be separate from the data that it operates on. This allows developers to write applications that only focus on the access and manipulation of data, and not the complications and overhead associated with how to actually perform these operations. IMS was later followed by the pioneering work in the early 1970s on the first relational DBMSs, IBM's System R and the University of California's INGRES. INGRES was soon adopted at other universities for their information systems and was subsequently commercialized in the late 1970s. Around the same time, Oracle released the first version of their DBMS that was similar to System R's design. Other companies were founded in the early 1980s that sought to repeat the success of the first commercial DBMSs, including Sybase and Informix. Although IBM never made System R available to the public, it later released a new relational DBMS (DB2) in 1983 that used parts of the System R code base.

The late 1980s and early 1990s brought about a new class of DBMSs that were designed to overcome the much touted impedance mismatch between the relational model and object-oriented programming languages [65]. These object-oriented DBMSs, however, never saw wide-spread market adoption because they lacked a standard interface like SQL. But many of the ideas from them were eventually incorporated in relational DBMSs when the major vendors added object and XML support a decade later, and then again in document-oriented NoSQL systems over two decades later.

The other notable event during the 1990s was the start of today's two major open-source DBMS projects. MySQL was started in Sweden in 1995 based on the earlier ISAM-based mSQL system. PostgreSQL began in 1994 when two Berkeley graduate students forked the original QUEL-based Postgres code from the 1980s to add support for SQL.

The 2000s brought the arrival of Internet applications that had more challenging resource requirements than applications from previous years. They needed to scale to support large number of concurrent users and had to be on-line all the time. But the database for these new applications was consistently found to be a bottleneck because the resource demands were much greater than what DBMSs and hardware could support at the time. Many tried the most obvious option of scaling their DBMS vertically by moving the database to a machine with better hardware. This, however, only improves performance so much and has diminishing returns. Furthermore, moving the database from one machine to another is a complex process and often requires significant downtime, which is unacceptable for these Web-based applications. To overcome this problem, some companies created custom *middleware* to shard single-node DBMSs over a cluster of less expensive machines. Such middleware presents a single logical database to the application that is stored across multiple physical nodes. When the application issues queries against this database, the middleware redirects and/or rewrites them to distribute their execution on one or more nodes in the cluster. The nodes execute these queries and send the results back to the middleware, which then coalesces them into a single response to the application. Two notable examples of this middleware approach were eBay's Oracle-based cluster [53] and Google's MySQL-based cluster [54]. This approach was later adopted by Facebook for their own MySQL cluster that is still used today.

Sharding middleware works well for simple operations like reading or updating a single record. It is more difficult, however, to execute queries that update more than one record in a transaction or join tables. As such, these early middleware

systems did not support these types of operations. eBay's middleware in 2002, for example, required their developers to implement all join operations in application-level code.

Eventually some of these companies moved away from using middleware and developed their own distributed DBMSs. The motivation for this was three-fold. Foremost was that traditional DBMSs at that time were focused on consistency and correctness at the expense of availability and performance. But this trade-off was deemed inappropriate for Web-based applications that need to be on-line all the time and have to support a large number of concurrent operations. Secondly, it was thought that there was too much overhead in using a full-featured DBMS like MySQL as a "dumb" data store. Likewise, it was also thought that the relational model was not the best way to represent an application's data and that using SQL was an overkill for simple look-up queries.

These problems turned out to be the origin of the impetus for the *NoSQL*[1] movement in the mid to late 2000s [22]. The key aspect of these NoSQL systems is that they forgo strong transactional guarantees and the relational model of traditional DBMSs in favor of eventual consistency and alternative data models (e.g., key/value, graphs, documents). This is because it was believed that these aspects of existing DBMSs inhibit their ability to scale out and achieve the high availability that is needed to support Web-based applications. The two most well-known systems that first followed this creed are Google's BigTable [23] and Amazon's Dynamo [26]. Neither of these two systems were available outside of their respective company at first (although they are now as cloud services), thus other organizations created their own open source clones of them. These include Facebook's Cassandra (based on BigTable and Dynamo) and PowerSet's Hbase (based on BigTable). Other start-ups created their own systems that were not necessarily copies of Google's or Amazon's systems but still followed the tenets of the NoSQL philosophy; the most well-known of these is MongoDB.

By the end of the 2000s, there was now a diverse set of scalable and more affordable distributed DBMSs available. The advantage of using a NoSQL system (or so people thought) was that developers could focus on the aspects of their application that were more beneficial to their business or organization, rather than having to worry about how to scale the DBMS. Many applications, however, are unable to use these NoSQL systems because they cannot give up strong transactional and consistency requirements. This is common for enterprise systems that handle high-profile data (e.g., financial and order processing systems). Some organizations, most notably Google [24], have found that NoSQL DBMSs cause their developers to spend too much time writing code to handle inconsistent data and that using transactions makes them more productive because they provide a useful abstraction that is easier for humans to reason about. Thus, the only options available for these organizations were to either purchase a more powerful single-node machine and to scale the DBMS vertically, or to develop their own custom sharding middleware that supports transactions. Both approaches are prohibitively expensive and are therefore not an option for many. It is in this environment that brought about NewSQL systems.

[1]The NoSQL community argues that the sobriquet should now be interpreted as "Not Only SQL", since some of these systems have since support some dialect of SQL.

2. THE RISE OF NEWSQL

Our definition of NewSQL is that they are a class of modern relational DBMSs that seek to provide the same scalable performance of NoSQL for OLTP read-write workloads while still maintaining ACID guarantees for transactions. In other words, these systems want to achieve the same scalability of NoSQL DBMSs from the 2000s, but still keep the relational model (with SQL) and transaction support of the legacy DBMSs from the 1970–80s. This enables applications to execute a large number of concurrent transactions to ingest new information and modify the state of the database using SQL (instead of a proprietary API). If an application uses a NewSQL DBMS, then developers do not have to write logic to deal with eventually consistent updates as they would in a NoSQL system. As we discuss below, this interpretation covers a number of both academic and commercial systems.

We note that there are data warehouse DBMSs that came out in the mid-2000s that some people think meet this criteria (e.g., Vertica, Greenplum, Aster Data). These DBMSs target on-line analytical processing (OLAP) workloads and should not be considered NewSQL systems. OLAP DBMSs are focused on executing complex read-only queries (i.e., aggregations, multi-way joins) that take a long time to process large data sets (e.g., seconds or even minutes). Each of these queries can be significantly different than the previous. The applications targeted by NewSQL DBMSs, on the other hand, are characterized as executing read-write transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index lookups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., executing the same queries with different inputs). Others have argued for a more narrow definition where a NewSQL system's implementation has to use (1) a lock-free concurrency control scheme and (2) a shared-nothing distributed architecture [57]. All of the DBMSs that we classify as NewSQL in Section 3 indeed share these properties and thus we agree with this assessment.

3. CATEGORIZATION

Given the above definition, we now examine the landscape of today's NewSQL DBMSs. To simplify this analysis, we will group systems based on the salient aspects of their implementation. The three categories that we believe best represent NewSQL systems are (1) novel systems that are built from the ground-up using a new architecture, (2) middleware that re-implement the same sharding infrastructure that was developed in the 2000s by Google and others, and (3) database-as-a-service offerings from cloud computing providers that are also based on new architectures.

Both authors have previously included alternative storage engines for existing single-node DBMSs in our categorization of NewSQL systems. The most common examples of these are replacements for MySQL's default InnoDB storage engine (e.g., TokuDB, ScaleDB, Akiban, deepSQL). The advantage of using a new engine is that an organization can get better performance without having to change anything in their application and still leverage the DBMS's existing ecosystem (e.g., tools, APIs). The most interesting of these was ScaleDB because it provided transparent sharding underneath the system without using middleware by redistributing execution between storage engines; the company, however, has since pivoted to another problem domain. There has been other sim-

ilar extensions for systems other than MySQL. Microsoft's in-memory Hekaton OLTP engine for SQL Server integrates almost seamlessly with the traditional, disk-resident tables. Others use Postgres' foreign data wrappers and API hooks to achieve the same type of integration but target OLAP workloads (e.g., Vitesse, CitusDB).

We now assert that such storage engines and extensions for single-node DBMSs are not representative of NewSQL systems and omit them from our taxonomy. MySQL's InnoDB has improved significantly in terms of reliability and performance, so the benefits of switching to another engine for OLTP applications are not that pronounced. We acknowledge that the benefits from switching from the row-oriented InnoDB engine to a column-store engine for OLAP workloads are more significant (e.g., Infobright, InfiniDB). But in general, the MySQL storage engine replacement business for OLTP workloads is the graveyard of failed database projects.

3.1 New Architectures

This category contains the most interesting NewSQL systems for us because they are new DBMSs built from scratch. That is, rather than extending an existing system (e.g., Microsoft's Hekaton for SQL Server), they are designed from a new codebase without any of the architectural baggage of legacy systems. All of the DBMSs in this category are based on distributed architectures that operate on shared-nothing resources and contain components to support multi-node concurrency control, fault tolerance through replication, flow control, and distributed query processing. The advantage of using a new DBMS that is built for distributed execution is that all parts of the system can be optimized for multi-node environments. This includes things like the query optimizer and communication protocol between nodes. For example, most NewSQL DBMSs are able to send intra-query data directly between nodes rather than having to route them to a central location like with some middleware systems.

Every one of the DBMSs in this category (with the exception of Google Spanner) also manages their own primary storage, either in-memory or on disk. This means that the DBMS is responsible for distributing the database across its resources with a custom engine instead of relying on an off-the-shelf distributed filesystem (e.g., HDFS) or storage fabric (e.g., Apache Ignite). This is an important aspect of them because it allows the DBMS to "send the query to the data" rather than "bring the data to the query," which results in significantly less network traffic since transmitting the queries is typically less network traffic than having to transmit data (not just tuples, but also indexes and materialized views) to the computation.

Managing their own storage also enables a DBMS to employ more sophisticated replication schemes than what is possible with the block-based replication scheme used in HDFS. In general, it allows these DBMSs to achieve better performance than other systems that are layered on top of other existing technologies; examples of this include the "SQL on Hadoop" systems like Trafodion [4] and Splice Machine [16] that provide transactions on top of Hbase. As such, we believe that such systems should not be considered NewSQL.

But there are downsides to using a DBMS based on a new architecture. Foremost is that many organizations are wary of adopting technologies that are too new and un-vetted with a large installation base. This means that the number of people that are experienced in the system is much smaller compared to the more popular DBMS vendors. It also means that an organization will potentially lose access to existing administration and reporting tools. Some DBMSs, like Clustrix and MemSQL, avoid this problem by maintaining compatibility with the MySQL wire protocol.

Examples: Clustrix [6], CockroachDB [7], Google Spanner [24], H-Store [8], HyPer [39], MemSQL [11], NuoDB [14], SAP HANA [55], VoltDB [17].

3.2 Transparent Sharding Middleware

There are now products available that provide the same kind of sharding middleware that eBay, Google, Facebook, and other companies developed in the 2000s. These allow an organization to split a database into multiple shards that are stored across a cluster of single-node DBMS instances. Sharding is different than database federation technologies of the 1990s because each node (1) runs the same DBMS, (2) only has a portion of the overall database, and (3) is not meant to be accessed and updated independently by separate applications.

The centralized middleware component routes queries, coordinates transactions, as well as manages data placement, replication, and partitioning across the nodes. There is typically a shim layer installed on each DBMS node that communicates with the middleware. This component is responsible for executing queries on behalf of the middleware at its local DBMS instance and returning results. All together, these allow middleware products to present a single logical database to the application without needing to modify the underlying DBMS.

The key advantage of using a sharding middleware is that they are often a drop-in replacement for an application that is already using an existing single-node DBMS. Developers do not need to make any changes to their application to use the new sharded database. The most common target for middleware systems is MySQL. This means that in order to be MySQL compatible, the middleware must support the MySQL wire protocol. Oracle provides the MySQL Proxy [13] and Fabric [12] toolkits to do this, but others have written their owning protocol handler library to avoid GPL licensing issues.

Although middleware makes it easy for an organization to scale their database out across multiple nodes, such systems still have to use a traditional DBMS on each node (e.g., MySQL, Postgres, Oracle). These DBMSs are based on the disk-oriented architecture that was developed in the 1970s, and thus they cannot use a storage manager or concurrency control scheme that is optimized for memory-oriented storage like in some of the NewSQL systems that are built on new architectures. Previous research has shown that the legacy components of disk-oriented architectures is a significant encumbrance that prevents these traditional DBMSs from scaling up to take advantage of higher CPU core counts and larger memory capacities [38]. The middleware approach can also incur redundant query planning and optimization on sharded nodes for complex queries (i.e., once at the middleware and once on the individual DBMS nodes), but this does allow each node to apply their own local optimizations on each query.

Examples: AgilData Scalable Cluster [2] [1], MariaDB MaxScale [10], ScaleArc [15], ScaleBase[3].

[2]Prior to 2015, AgilData Cluster was known as dbShards.

[3]ScaleBase was acquired by ScaleArc in 2015 and is no longer sold.

3.3 Database-as-a-Service

Lastly, there are cloud computing providers that offer NewSQL database-as-a-service (DBaaS) products. With these services, organizations do not have to maintain the DBMS on either their own private hardware or on a cloud-hosted virtual machine (VM). Instead, the DBaaS provider is responsible for maintaining the physical configuration of the database, including system tuning (e.g., buffer pool size), replication, and backups. The customer is provided with a connection URL to the DBMS, along with a dashboard or API to control the system.

DBaaS customers pay according to their expected application's resource utilization. Since database queries vary widely in how they use computing resources, DBaaS providers typically do not meter query invocations in the same way that they meter operations in block-oriented storage services (e.g., Amazon's S3, Google's Cloud Storage). Instead, customers subscribe to a pricing tier that specifies the maximum resource utilization threshold (e.g., storage size, computation power, memory allocation) that the provider will guarantee.

As in most aspects of cloud computing, the largest companies are the major players in the DBaaS field due to the economies of scale. But almost all of the DBaaSs just provide a managed instance of a traditional, single-node DBMS (e.g., MySQL): notable examples include Google Cloud SQL, Microsoft Azure SQL, Rackspace Cloud Database, and Salesforce Heroku. We do not consider these to be NewSQL systems as they use the same underlying disk-oriented DBMSs based on the 1970s architectures. Some vendors, like Microsoft, retro-fitted their DBMS to provide better support for multi-tenant deployments [21].

We instead regard only those DBaaS products that are based on a new architecture as NewSQL. The most notable examples is Amazon's Aurora for their MySQL RDS. Its distinguishing feature over InnoDB is that it uses a log-structured storage manager to improve I/O parallelism.

There are also companies that do not maintain their own data centers but rather sell DBaaS software that run on top of these public cloud platforms. ClearDB provides their own custom DBaaS that can be deployed on all of the major cloud platforms. This has the advantage that it can distribute a database across different providers in the same geographical region to avoid downtimes due to service outages.

Aurora and ClearDB are the only two products available in this NewSQL category as of 2016. We note that several companies in this space have failed (e.g., GenieDB, Xeround), forcing their customers to scramble to find a new provider and migrate their data out of those DBaaS before they were shut down. We attribute their failure due to being ahead of market demand and from being out-priced from the major vendors.

Examples: Amazon Aurora [3], ClearDB [5].

4. THE STATE OF THE ART

We next discuss the features of NewSQL DBMSs to understand what (if anything) is novel in these systems. A summary of our analysis is shown in Table 1.

4.1 Main Memory Storage

All of the major DBMSs use a disk-oriented storage architecture based on the original DBMSs from the 1970s. In these systems, the primary storage location of the database is assumed to be on a block-addressable durable storage device, like an SSD or HDD. Since reading and writing to these devices is slow, DBMSs use memory to cache blocks read from disk and to buffer updates from transactions. This was necessary because historically memory was much more expensive and had a limited capacity compared to disks. We have now reached the point, however, where capacities and prices are such that it is affordable to store all but the largest OLTP databases entirely in memory. The benefit of this approach is that it enables certain optimizations because the DBMS no longer has to assume that a transaction could access data at any time that is not in memory and will have to stall. Thus, these systems can get better performance because many of the components that are necessary to handle these cases, like a buffer pool manager or heavy-weight concurrency control schemes, are not needed [38].

There are several NewSQL DBMSs that are based on a main memory storage architecture, including both academic (e.g., H-Store, HyPer) and commercial (e.g., MemSQL, SAP HANA, VoltDB) systems. These systems perform significantly better than disk-based DBMSs for OLTP workloads because of this main memory orientation.

The idea of storing a database entirely in main memory is not a new one [28, 33]. The seminal research at the University of Wisconsin-Madison in the early 1980s established the foundation for many aspects of main memory DBMSs [43], including indexes, query processing, and recovery algorithms. In that same decade, the first distributed main-memory DBMSs, PRISMA/DB, was also developed [40]. The first commercial main memory DBMSs appeared in 1990s; Altibase [2], Oracle's TimesTen [60], and AT&T's DataBlitz [20] were early proponents of this approach.

One thing that is new with main memory NewSQL systems is the ability to evict a subset of the database out to persistent storage to reduce its memory footprint. This allows the DBMS to support databases that are larger than the amount of memory available without having to switch back to a disk-oriented architecture. The general approach is to use an internal tracking mechanism inside of the system to identify which tuples are not being accessed anymore and then chose them for eviction. H-Store's *anti-caching* component moves cold tuples to a disk-resident store and then installs a "tombstone" record in the database with the location of the original data [25]. When a transaction tries to access a tuple through one of these tombstones, it is aborted and then a separate thread asynchronously retrieves that record and moves it back into memory. Another variant for supporting larger-than-memory databases is an academic project from EPFL that uses OS virtual memory paging in VoltDB [56]. To avoid false negatives, all of these DBMSs retain the keys for evicted tuples in databases' indexes, which inhibits the potential memory savings for those applications with many secondary indexes. Although not a NewSQL DBMS, Microsoft's *Project Siberia* [29] for Hekaton maintains a Bloom filter per index to reduce the in-memory storage overhead of tracking evicted tuples.

Another DBMS that takes a different approach for larger-than-memory databases is MemSQL where an administrator can manually instruct the DBMS to store a table in a columnar format. MemSQL does not maintain any in-memory tracking meta-data for these disk-resident tuples. It organizes this data in log-structured storage to reduce the overhead of updates,

which are traditionally slow in OLAP data warehouses.

4.2 Partitioning / Sharding

The way that almost all of the distributed NewSQL DBMSs scale out is to split a database up into disjoint subsets, called either partitions or shards.

Distributed transaction processing on partitioned databases is not a new idea. Many of the fundamentals of these systems came from the seminal work by the great Phil Bernstein (and others) in the SDD-1 project in the late 1970s [51]. In the early 1980s, the teams behind the two pioneering, single-node DBMSs, System R and INGRES, both also created distributed versions of their respective systems. IBM's R* was a shared-nothing, disk-oriented distributed DBMS like SDD-1 [63]. The distributed version of INGRES is mostly remembered for its dynamic query optimization algorithm that recursively breaks a distributed query into smaller pieces [31]. Later, the GAMMA project [27] from the University of Wisconsin-Madison explored different partitioning strategies.

But these earlier distributed DBMSs never caught on for two reasons. The first of these was that computing hardware in the 20th century was so expensive that most organizations could not afford to deploy their database on a cluster of machines. The second issue was that the application demand for a high-performance distributed DBMS was simply not there. Back then the expected peak throughput of a DBMS was typically measured at tens to hundreds of transactions per second. We now live in an era where both of these assumptions are no longer true. Creating a large-scale, data-intensive application is easier now than it ever has been, in part due to the proliferation of open-source distributed system tools, cloud computing platforms, and affordable mobile devices.

The database's tables are horizontally divided into multiple fragments whose boundaries are based on the values of one (or more) of the table's columns (i.e., the partitioning attributes). The DBMS assigns each tuple to a fragment based on the values of these attributes using either range or hash partitioning. Related fragments from multiple tables are combined together to form a partition that is managed by a single node. That node is responsible for executing any query that needs to access data stored in its partition. Only the DBaaS systems (Amazon Aurora, ClearDB) do not support this type of partitioning.

Ideally, the DBMS should be able to also distribute the execution of a query to multiple partitions and then combine their results together into a single result. All of the NewSQL systems except for ScaleArc that support native partitionining provide this functionality.

The databases for many OLTP applications have a key property that makes them amenable to partitioning. Their database schemas can be transposed into a tree-like structure where descendants in the tree have a foreign key relationship to the root [58]. The tables are then partitioned on the attributes involved in these relationships such that all of the data for a single entity are co-located together in the same partition. For example, the root of the tree could be the customer table, and the database is partitioned such that each customer, along with their order records and account information, are stored together. The benefit of this is that it allows most (if not all) transactions to only need to access data at a single partition. This in turn reduces the communication overhead of the system because it does not have to use an atomic commitment protocol (e.g., two-phase commit) to make sure that transactions finish correctly at different nodes.

The NewSQL DBMSs that deviate from the homegenous cluster node architecture are NuoDB and MemSQL. For NuoDB, it designates one or more nodes as storage managers (SM) that each store a partition of the database. The SMs splits a database into blocks (called "atoms" in NuoDB parlance). All other nodes in the cluster are designated as transaction engines (TEs) that act as an in-memory cache of atoms. To process a query, a TE node retrieves all of the atoms that it needs for that query (either from the appropriate SMs or from other TEs). TEs acquire write-locks on tuples and then broadcasts any changes to atoms to the other TEs and the SM. To avoid atoms from moving back and forth between nodes, NuoDB exposes load-balancing schemes to ensure that data that is used together often reside at the same TE. This means that NuoDB ends up with the same partitioning scheme as the other distributed DBMSs but without having to pre-partition the database or identify the relationships between tables.

MemSQL also uses a similar heterogeneous architecture comprised of execution-only aggregator nodes and leaf nodes that store the actual data. The difference between these two systems is in how they reduce the amount of data that is pulled from the storage nodes to the execution nodes. With NuoDB, the TEs cache atoms to reduce the amount data that they read from the SMs. MemSQL's aggregator nodes do not cache any data, but the leaf nodes execute parts of queries to reduce the amount of data that is sent to the aggregator nodes; this is not possible in NuoDB because the SMs are only a data store.

These two systems are able to add additional execution resources to the DBMS's cluster (NuoDB's TE nodes, MemSQL's aggregator nodes) without needing to re-partition the database. A research prototype of SAP HANA also explored using this approach [36]. It remains to be seen, however, whether such a heterogeneous architecture is superior to a homegenous one (i.e., were each node both stores data and executes queries) in terms of either performance or operational complexity.

Another aspect of partitioning in NewSQL systems that is new is that some of them support live migration. This allows the DBMS to move data between physical resources to re-balance and alleviate hotspots, or to increase/decrease the DBMS's capacity without any interruption to service. This is similar to re-balancing in NoSQL systems, but it is more difficult because a NewSQL DBMS has to maintain ACID guarantees for transactions during the migration [30]. There two approaches that DBMSs use to achieve this. The first is to organize the database in many coarse-grained "virtual" (i.e., logical) partitions that are spread amongst the physical nodes [52]. Then when the DBMS needs to re-balance, it moves these virtual partitions between nodes. This is the approach used in Clustrix and AgilData, as well as in NoSQL systems like Cassandra and DynamoDB. The other approach is for the DBMS to perform more fine-grained re-balancing by redistributing individual tuples or groups of tuples through range partitioning. This is akin to the auto-sharding feature in the MongoDB NoSQL DBMS. It is used in systems like ScaleBase and H-Store [30].

4.3 Concurrency Control

Concurrency control scheme is the most salient and important implementation detail of a transaction processing DBMS as it affects almost all aspects of the system. Concurrency control permits end-users to access a database in a multi-program-

med fashion while preserving the illusion that each of them is executing their transaction alone on a dedicated system. It essentially provides the atomicity and isolation guarantees in the system, and as such it influences the entire system's behavior.

Beyond which scheme a system uses, another important aspect of the design of a distributed DBMS is whether the system uses a centralized or decentralized transaction coordination protocol. In a system with a centralized coordinator, all transactions' operations have to go through the coordinator, which then makes decisions about whether transactions are allowed to proceed or not. This is the same approach used by the TP monitors of the 1970–1980s (e.g., IBM CICS, Oracle Tuxedo). In a decentralized system, each node maintains the state of transactions that access the data that it manages. The nodes then have to coordinate with each other to determine whether concurrent transactions conflict. A decentralized coordinator is better for scalability but requires that the clocks in the DBMS nodes are highly synchronized in order to generate a global ordering of transactions [24].

The first distributed DBMSs from the 1970–80s used two-phase locking (2PL) schemes. SDD-1 was the first DBMS specifically designed for distributed transaction processing across a cluster of shared-nothing nodes managed by a centralized coordinator. IBM's R* was similar to SDD-1, but the main difference was that the coordination of transactions in R* was completely decentralized; it used distributed 2PL protocol where transactions locked data items that they access directly at nodes. The distributed version of INGRES also used decentralized 2PL with centralized deadlock detection.

Almost all of the NewSQL systems based on new architectures eschew 2PL because the complexity of dealing with deadlocks. Instead, the current trend is to use variants of timestamp ordering (TO) concurrency control where the DBMS assumes that transactions will not execute interleaved operations that will violate serializable ordering. The most widely used protocol in NewSQL systems is decentralized multi-version concurrency control (MVCC) where the DBMS creates a new version of a tuple in the database when it is updated by a transaction. Maintaining multiple versions potentially allows transactions to still complete even if another transaction updates the same tuples. It also allows for long-running, read-only transactions to not block on writers. This protocol is used in almost all of the NewSQL systems based on new architectures, like MemSQL, HyPer, HANA, and CockroachDB. Although there are engineering optimizations and tweaks that these systems use in their MVCC implementations to improve performance, the basic concepts of the scheme are not new. The first known work describing MVCC is a MIT PhD dissertation from 1979 [49], while the first commercial DBMSs to use it were Digital's VAX Rdb and InterBase in the early 1980s. We note that the architecture of InterBase was designed by Jim Starkey, who is also the original designer of NuoDB and the failed Falcon MySQL storage engine project.

Other systems use a combination of 2PL and MVCC together. With this approach, transactions still have to acquire locks under the 2PL scheme to modify the database. When a transaction modifies a record, the DBMS creates a new version of that record just as it would with MVCC. This scheme allows read-only queries to avoid having to acquire locks and therefore not block on writing transactions. The most famous implementation of this approach is MySQL's InnoDB, but it is also used in both Google's Spanner, NuoDB, and Clustrix. NuoDB improves on the original MVCC by employing a gossip protocol to broadcast versioning information between nodes.

All of the middleware and DBaaS services inherit the concurrency control scheme of their underlying DBMS architecture; since most of them use MySQL, this makes them 2PL with MVCC systems.

We regard the concurrency control implementation in Spanner (along with its descendants F1 [54] and SpannerSQL) as one of the most novel of the NewSQL systems. The actual scheme itself is based on the 2PL and MVCC combination developed in previous decades. But what makes Spanner different is that it uses hardware devices (e.g., GPS, atomic clocks) for high-precision clock synchronization. The DBMS uses these clocks to assign timestamps to transactions to enforce consistent views of its multi-version database over wide-area networks. CockroachDB also purports to provide the same kind of consistency for transactions across data centers as Spanner but without the use of atomic clocks. They instead rely on a hybrid clock protocol that combines loosely synchronized hardware clocks and logical counters [41].

Spanner is also noteworthy because it heralds Google's return to using transactions for its most critical services. The authors of Spanner even remark that it is better to have their application programmers deal with performance problems due to overuse of transactions, rather than writing code to deal with the lack of transactions as one does with a NoSQL DBMS [24].

Lastly, the only commercial NewSQL DBMS that is not using some MVCC variant is VoltDB. This system still uses TO concurrency control, but instead of interleaving transactions like in MVCC, it schedules transactions to execute one-at-a-time at each partition. It also uses a hybrid architecture where single-partition transactions are scheduled in a decentralized manner but multi-partition transactions are scheduled with a centralized coordinator. VoltDB orders transactions based on logical timestamps and then schedules them for execution at a partition when it is their turn. When a transaction executes at a partition, it has exclusive access to all of the data at that partition and thus the system does not have to set fine-grained locks and latches on its data structures. This allows transactions that only have to access a single partition to execute efficiently because there is no contention from other transactions. The downside of partition-based concurrency control is that it does not work well if transactions span multiple partitions because the network communication delays cause nodes to sit idle while they wait for messages. This partition-based concurrency is not a new idea. An early variant of it was first proposed in a 1992 paper by Hector Garcia-Molina [34] and implemented in the kdb system in late 1990s [62] and in H-Store (which is the academic predecessor of VoltDB).

In general, we find that there is nothing significantly new about the core concurrency control schemes in NewSQL systems other than laudable engineering to make these algorithms work well in the context of modern hardware and distributed operating environments.

4.4 Secondary Indexes

A secondary index contains a subset of attributes from a table that are different than its primary key(s). This allows the DBMS to support fast queries beyond primary key or partitioning key look-ups. They are trivial to support in a non-partitioned DBMS because the entire database is located on a

single node. The challenge with secondary indexes in a distributed DBMS is that they cannot always be partitioned in the same manner as with the rest of the database. For example, suppose that the tables of a database are partitioned based on the customer's table primary key. But then there are some queries that want to do a reverse look-up from the customer's email address to the account. Since the tables are partitioned on the primary key, the DBMS will have to broadcast these queries to every node, which is obviously inefficient.

The two design decisions for supporting secondary indexes in a distributed DBMS are (1) where the system will store them and (2) how it will maintain them in the context of transactions. In a system with a centralized coordinator, like with sharding middleware, secondary indexes can reside on both the coordinator node and the shard nodes. The advantage of this approach is that there is only a single version of the index in the entire system, and thus it is easier to maintain.

All of the NewSQL systems based on new architectures are decentralized and use partitioned secondary indexes. This means that each node stores a portion of the index, rather than each node having a complete copy of it. The trade-off between partitioned and replicated indexes is that with the former queries may need to span multiple nodes to find what they are looking for but if a transaction updates an index it will only have to modify one node. In a replicated index, the roles are reversed: a look-up query can be satisfied by just one node in the cluster, but any time a transaction modifies the attributes referenced in secondary index's underlying table (i.e., the key or the value), the DBMS has to execute a distributed transaction that updates all copies of the index.

An example of a decentralized secondary index that mixes both of these concepts is in Clustrix. The DBMS first maintains a replicated, coarse-grained (i.e., range-based) index at each node that maps values to partitions. This mapping allows the DBMS to route queries to the appropriate node using an attribute that is not the table's partitioning attribute. These queries will then access a second partitioned index at that node that maps exact values to tuples. Such a two-tier approach reduces the amount of coordination that is needed to keep the replicated index in sync across the cluster since it only maps ranges instead of individual values.

The most common way that developers create secondary indexes when using a NewSQL DBMS that does not support them is to deploy an index using an in-memory, distributed cache, such as Memcached [32]. But using an external system requires the application to maintain the cache since the DBMSs will not automatically invalidate the external cache.

4.5 Replication

The best way that an organization can ensure high availability and data durability for their OLTP application is to replicate their database. All modern DBMSs, including NewSQL systems, support some kind of replication mechanism. DBaaS have a distinct advantage in this area because they hide all of the gritty details of setting of replication from their customers. They make it easy to deploy a replicated DBMS without the administrator having to worry about transmitting logs and making sure that nodes are in sync.

There are two design decisions when it comes to database replication. The first is how the DBMS enforces data consistency across nodes. In a *strongly consistent* DBMS, a transaction's writes must be acknowledged and installed at all replicas before that transaction is considered committed (i.e., durable). The advantage of this approach is that replicas can serve read-only queries and still be consistent. That is, if the application receives an acknowledgement that a transaction has committed, then any modifications made by that transaction are visible to any subsequent transaction in the future regardless of what DBMS node they access. It also means that when a replica fails, there are no lost updates because all the other nodes are synchronized. But maintaining this synchronization requires the DBMS to use an atomic commitment protocol (e..g, two-phase commit) to ensure that all replicas agree with the outcome of a transaction, which has additional overhead and can lead to stalls if a node fails or if there is a network partition/delay. This is why NoSQL systems opt for a *weakly consistent* model (also called eventual consistency) where not all replicas have to acknowledge a modification before the DBMS notifies the application that the write succeeded.

All of the NewSQL systems that we are aware of support strongly consistent replication. But there is nothing novel about how these systems ensure this consistency. The fundamentals of state machine replication for DBMSs were studied back in the 1970s [37, 42]. NonStop SQL was one of the first distributed DBMSs built in the 1980s using strongly consistency replication to provide fault tolerance in this same manner [59].

In addition to the policy of when a DBMS propagates updates to replicas, there are also two different execution models for how the DBMS performs this propagation. The first, known as *active-active* replication, is where each replica node processes the same request simultaneously. For example, when a transaction executes a query, the DBMS executes that query in parallel at all of the replicas. This is different from *active-passive* replication where a request is first processed at a single node and then the DBMS transfers the resultant state to the other replicas. Most NewSQL DBMSs implement this second approach because they use a non-deterministic concurrency control scheme. This means that they cannot send queries to replicas as they arrive on the master because they may get executed in a different order on the replicas and the state of the databases will diverge at each replica. This is because their execution order depends on several factors, including network delays, cache stalls, and clock skew.

Deterministic DBMSs (e.g., H-Store, VoltDB, ClearDB) on the other hand do not perform these additional coordination steps. This is because the DBMS guarantees that transactions' operations execute in the same order on each replica and thus the state of the database is guaranteed to be the same [44]. Both VoltDB and ClearDB also ensure that the application does not execute queries that utilize sources of information that are external to the DBMS that may be different on each replica (e.g., setting a timestamp field to the local system clock).

One aspect of the NewSQL systems that is different than previous work outside of academia is the consideration of replication over the wide-area network (WAN). This is a byproduct of modern operating environments where it is now trivial to deploy systems across multiple data centers that are separated by large geographical differences. Any NewSQL DBMS can be configured to provide synchronous updates of data over the WAN, but this would cause significant slowdown for normal operations. Thus, they instead provide asynchronous replication methods. To the best of our knowledge, Spanner and CockroachDB are the only NewSQL systems to provide a repli-

cation scheme that is optimized for strongly consistent replicas over the WAN. They again achieve this through a combination of atomic and GPS hardware clocks (in case of Spanner [24]), or hybrid clocks (in the case of CockroachDB [41]).

4.6 Crash Recovery

Another important feature of a NewSQL DBMS for providing fault tolerance is its crash recovery mechanism. But unlike traditional DBMSs where the main concern of fault tolerance is to ensure that no updates are lost [47], newer DBMSs must also minimize downtime. Modern web applications are expected to be on-line all the time and site outages are costly.

The traditional approach to recovery in a single-node system without replicas is that when the DBMS comes back online after a crash, it loads in the last checkpoint that it took from disk and then replays its write-ahead log (WAL) to return the state of the database to where it was at the moment of the crash. The canonical method of this approach, known as ARIES [47], was invented by IBM researchers in the 1990s. All major DBMSs implement some variant of ARIES.

In a distributed DBMS with replicas, however, the traditional single-node approach is not directly applicable. This is because when the master node crashes, the system will promote one of the slave nodes to be the new master. When the previous master comes back on-line, it cannot just load in its last checkpoint and rerun its WAL because the DBMS has continued to process transactions and therefore the state of the database has moved forward. The recovering node needs to get the updates from the new master (and potentially other replicas) that it missed while it was down. There are two potential ways to do this. The first is for the recovering node to load in its last checkpoint and WAL from its local storage and then pull log entries that it missed from the other nodes. As long as the node can process the log faster than new updates are appended to it, the node will eventually converge to the same state as the other replica nodes. This is possible if the DBMS uses physical or physiological logging, since the time to apply the log updates directly to tuples is much less than the time it takes to execute the original SQL statement. To reduce the time it takes to recover, the other option is for the recovering node to discard its checkpoint and have system take a new one that the node will recover from. One additional benefit of this approach is that this same mechanism can also be used in the DBMS to add a new replica node.

The middleware and DBaaS systems rely on the built-in mechanisms of their underlying single-node DBMSs, but add additional infrastructure for leader election and other management capabilities. The NewSQL systems that are based on new architectures use a combination of off-the-shelf components (e.g., ZooKeeper, Raft) and their own custom implementations of existing algorithms (e.g., Paxos). All of these are standard procedures and technologies that have been available in commercial distributed systems since the 1990s.

5. FUTURE TRENDS

We foresee the next trend for database applications in the near future is the ability to execute analytical queries and machine learning algorithms on freshly obtained data. Such workloads, colloquially known as "real-time analytics" or hybrid transaction-analytical processing (HTAP), seek to extrapolate insights and knowledge by analyzing a combination of historical data sets with new data [35]. This differs from traditional business intelligence operations from the previous decade that could only perform this analysis on historical data. Having a shorter turnaround time is important in modern applications because data has immense value as soon as it is created, but that value diminishes over time.

There are three approaches to supporting HTAP pipelines in a database application. The most common is to deploy separate DBMSs: one for transactions and another for analytical queries. With this architecture, the front-end OLTP DBMS stores all of the new information generated from transactions. Then in the background, the system uses an extract-transform-load utility to migrate data from this OLTP DBMS to a second back-end data warehouse DBMS. The application executes all complex OLAP queries in the back-end DBMS to avoid slowing down the OLTP system. Any new information generated from the OLAP system is pushed forward to front-end DBMS.

Another prevailing system design, known as the lambda architecture [45], is to use a separate batch processing system (e.g., Hadoop, Spark) to compute a comprehensive view on historical data, while simultaneously using a stream processing system (e.g., Storm [61], Spark Streaming [64]) to provide views of incoming data. In this split architecture, the batch processing system periodically rescans the data set and performs a bulk upload of the result to the stream processing system, which then makes modifications based on new updates.

There are several problems inherent with the bifurcated environment of these two approaches. Foremost is that the time it takes to propagate changes between the separate systems is often measured in minutes or even hours. This data transfer inhibits an application's ability to act on data immediately when it is entered in the database. Second, the administrative overhead of deploying and maintaining two different DBMSs is non-trivial as personnel is estimated to be almost 50% of the total ownership cost of a large-scale database system [50]. It also requires the application developer to write a query for multiple systems if they want to combine data from different databases. Some systems that try to achieve a single platform by hiding this split system architecture; an example of this is Splice Machine [16], but this approach has other technical issues due to copying data from the OLTP system (Hbase) before it can be used in the OLAP system (Spark).

The third (and in our opinion better) approach is to use a single HTAP DBMS that supports the high throughput and low latency demands of OLTP workloads, while also allowing for complex, longer running OLAP queries to operate on both hot (transactional) and cold (historical) data. What makes these newer HTAP systems different from legacy general-purpose DBMSs is that they incorporate the advancements from the last decade in the specialized OLTP (e.g., in-memory storage, lock-free execution) and OLAP (e.g., columnar storage, vectorized execution) systems, but within a single DBMS.

SAP HANA and MemSQL were the first NewSQL DBMSs to market themselves as HTAP systems. HANA achieves this by using multiple execution engines internally: one engine for row-oriented data that is better for transactions and a different engine for column-oriented data that is better for analytical queries. MemSQL uses two different storage managers (one for rows, one for columns) but mixes them together in a single execution engine. HyPer switched from a row-oriented system with H-Store-style concurrency control that was focused on

		Year Released	Main Memory Storage	Partitioning	Concurrency Control	Replication	Summary
NEW ARCHITECTURES	**Clustrix** [6]	2006	No	Yes	MVCC+2PL	Strong+Passive	MySQL-compatible DBMS that supports shared-nothing, distributed execution.
	CockroachDB [7]	2014	No	Yes	MVCC	Strong+Passive	Built on top of distributed key/value store. Uses software hybrid clocks for WAN replication.
	Google Spanner [24]	2012	No	Yes	MVCC+2PL	Strong+Passive	WAN-replicated, shared-nothing DBMS that uses special hardware for timestamp generation.
	H-Store [8]	2007	Yes	Yes	TO	Strong+Active	Single-threaded execution engines per partition. Optimized for stored procedures.
	HyPer [9]	2010	Yes	Yes	MVCC	Strong+Passive	HTAP DBMS that uses query compilation and memory efficient indexes.
	MemSQL [11]	2012	Yes	Yes	MVCC	Strong+Passive	Distributed, shared-nothing DBMS using compiled queries. Supports MySQL wire protocol.
	NuoDB [14]	2013	Yes	Yes	MVCC	Strong+Passive	Split architecture with multiple in-memory executor nodes and a single shared storage node.
	SAP HANA [55]	2010	Yes	Yes	MVCC	Strong+Passive	Hybrid storage (rows + cols). Amalgamation of previous TREX, P*TIME, and MaxDB systems.
	VoltDB [17]	2008	Yes	Yes	TO	Strong+Active	Single-threaded execution engines per partition. Supports streaming operators.
MIDDLEWARE	**AgilData** [1]	2007	No	Yes	MVCC+2PL	Strong+Passive	Shared-nothing database sharding over single-node MySQL instances.
	MariaDB MaxScale [10]	2015	No	Yes	MVCC+2PL	Strong+Passive	Query router that supports custom SQL rewriting. Relies on MySQL Cluster for coordination.
	ScaleArc [15]	2009	No	Yes	Mixed	Strong+Passive	Rule-based query router for MySQL, SQL Server, and Oracle.
DBAAS	**Amazon Aurora** [3]	2014	No	No	MVCC	Strong+Passive	Custom log-structured MySQL engine for RDS.
	ClearDB [5]	2010	No	No	MVCC+2PL	Strong+Active	Centralized router that mirrors a single-node MySQL instance in multiple data centers.

Table 1: NewSQL Systems – Summary of the system features described in Section 4 for the different DBMSs. Note that the year released is either when the project was announced publicly or when the company was first formed.

OLTP to use an HTAP column-store architecture with MVCC to allow it support more complex OLAP queries [48]. Even VoltDB has pivoted their marketing strategy from pure OLTP performance to providing streaming semantics. Similarly, the S-Store project seeks to add support for stream processing operations on top of the H-Store architecture [46]. It is likely that the specialized OLAP systems from the mid-2000s (e.g., Greenplum) will start to add support for better OLTP.

We note, however, that the rise of HTAP DBMSs does mean the end of giant, monolithic OLAP warehouses. Such systems will still be necessary in the short-term as they stand to be the universal back-end database for all of an organization's front-end OLTP silos. But eventually the resurgence of database federation will allow organization's to execute analytical queries that span multiple OLTP databases (including even multiple vendors) without needing to move data around.

6. CONCLUSION

The main takeaway from our analysis is that NewSQL database systems are not a radical departure from existing system architectures but rather represent the next chapter in the continuous development of database technologies. Most of the techniques that these systems employ have existed in previous DBMSs from academia and industry. But many of them were only implemented one-at-a-time in a single system and never all together. What is therefore innovative about these NewSQL DBMSs is that they incorporate these ideas into single platforms. Achieving this is by no means a trivial engineering effort. They are by-products of a new era where distributed computing resources are plentiful and affordable, but at the same time the demands of applications is much greater.

It is also interesting to consider the potential impact and future direction of NewSQL DBMSs in the marketplace. Given that the legacy DBMS vendors are entrenched and well funded, NewSQL systems have an uphill battle to gain market share. In the last five years since we first coined the term NewSQL [18], several NewSQL companies have folded (e.g., GenieDB, Xeround, Translattice) or pivoted to focus on other problem domains (e.g., ScaleBase, ParElastic). Based on our analysis and interviews with several companies, we have found that NewSQL systems have had a relatively slow rate of adoption, especially compared to the developer-driven NoSQL uptake. This is because NewSQL DBMSs are designed to support the transactional workloads that are mostly found in enterprise applications. Decisions regarding database choices for these enterprise applications are likely to be more conservative than for new Web application workloads. This is also evident from the fact that we find that NewSQL DBMSs are used to complement or replace existing RDBMS deployments, whereas NoSQL are being deployed in new application workloads [19].

Unlike with the OLAP DBMS start-ups from the 2000s, where almost all of the vendors were acquired by major technology companies, up until now there has been only one acquisition made of a NewSQL company. In March 2016, Tableau announced that it purchased the start-up formed for the HyPer project. The two other possible exceptions to this are (1) Apple acquiring FoundationDB in March 2015, but we exclude them because this system was at its core a NoSQL key-value store with an inefficient SQL layer grafted on top of it, and (2) ScaleArc acquiring ScaleBase, but this was one competitor buying out another. None of these examples are the same kind of acquisition where a legacy vendor purchasing an upstart

system (e.g., Teradata buying Aster Data Systems in 2011). We instead see that the large vendors are choosing to innovate and improve their own systems rather than acquire NewSQL start-ups. Microsoft added the in-memory Hekaton engine to SQL Server in 2014 to improve OLTP workloads. Oracle and IBM have been slightly slower to innovate; they recently added column-oriented storage extensions to their systems to compete with the rising popularity of OLAP DBMSs like HP Vertica and Amazon Redshift. It is possible that they will add an in-memory option for OLTP workloads in the future.

More long term, we believe that there will be a convergence of features in the four classes of systems that we discussed here: (1) the older DBMSs from the 1980-1990s, (2) the OLAP data warehouses from the 2000s, (3) the NoSQL DBMSs from the 2000s, and (4) the NewSQL DBMSs from the 2010s. We expect that all of the key systems in these groups will support some form of the relational model and SQL (if they do not already), as well as both OLTP operations and OLAP queries together like HTAP DBMSs. When this occurs, such labels will be meaningless.

ACKNOWLEDGMENTS

The authors would like to thank the following people for their feedback: Andy Grove (AgilData), Prakhar Verma (Amazon), Cashton Coleman (ClearDB), Dave Anselmi (Clustrix), Spencer Kimball (CockroachDB), Peter Mattis (CockroachDB), Ankur Goyal (MemSQL), Seth Proctor (NuoDB), Anil Goel (SAP HANA), Ryan Betts (VoltDB). This work was supported (in part) by the National Science Foundation (Award CCF-1438955).

For questions or comments about this paper, please call the CMU Database Hotline at +1-844-88-CMUDB.

7. REFERENCES

[1] AgilData Scalable Cluster for MySQL. http://www.agildata.com/.
[2] Altibase. http://altibase.com.
[3] Amazon Aurora. https://aws.amazon.com/rds/aurora.
[4] Apache Trafodion. http://trafodion.apache.org.
[5] ClearDB. https://www.cleardb.com.
[6] Clustrix. http://www.clustrix.com.
[7] CockroachDB. https://www.cockroachlabs.com/.
[8] H-Store. http://hstore.cs.brown.edu.
[9] HyPer. http://hyper-db.de.
[10] MariaDB MaxScale. https://mariadb.com/products/mariadb-maxscale.
[11] MemSQL. http://www.memsql.com.
[12] MySQL Fabric. https://www.mysql.com/products/enterprise/fabric.html.
[13] MySQL Proxy. http://dev.mysql.com/doc/mysql-proxy/en/.
[14] NuoDB. http://www.nuodb.com.
[15] ScaleArc. http://scalearc.com.
[16] Splice Machine. http://www.splicemachine.com.
[17] VoltDB. http://www.voltdb.com.
[18] M. Aslett. How will the database incumbents respond to NoSQL and NewSQL? The 451 Group, April 2011.
[19] M. Aslett. MySQL vs. NoSQL and NewSQL: 2011-2015. The 451 Group, May 2012.
[20] J. Baulier, P. Bohannon, S. Gogate, S. Joshi, C. Gupta, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, A. Silberschatz, and S. Sudarshan. DataBlitz: A high performance main-memory storage manager. VLDB, pages 701–, 1998.
[21] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft SQL server for cloud computing. In *ICDE*, pages 1255–1263, 2011.
[22] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39:12–27, 2011.
[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.
[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.
[25] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.
[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.
[27] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *VLDB*, pages 228–237, 1986.
[28] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.
[29] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment*, 7(11):931–942, 2014.
[30] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. E. Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. SIGMOD, pages 299–313, 2015.
[31] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. SIGMOD, pages 169–180, 1978.
[32] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004.
[33] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *IEEE Trans. Comput.*, 33(5):391–399, May 1984.
[34] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, Dec. 1992.

[35] Gartner. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/doc/2657815/, 2014.

[36] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, Aug. 2015.

[37] J. Gray. *Concurrency Control and Recovery in Database Systems*, chapter Notes on data base operating systems, pages 393–481. Springer-Verlag, 1978.

[38] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[39] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE, pages 195–206, 2011.

[40] M. L. Kersten, P. M. Apers, M. A. Houtsma, E. J. Kuyk, and R. L. Weg. A distributed, main-memory database machine. In *Database Machines and Knowledge Base Machines*, volume 43 of *The Kluwer International Series in Engineering and Computer Science*, pages 353–369. 1988.

[41] S. Kimball. Living without atomic clocks. https://www.cockroachlabs.com/blog/living-without-atomic-clocks/, February 2016.

[42] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[43] T. J. Lehman. *Design and performance evaluation of a main memory relational database system*. PhD thesis, University of Wisconsin–Madison, 1986.

[44] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *ICDE*, pages 604–615, 2014.

[45] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications, 2013.

[46] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-store: Streaming meets transaction processing. *PVLDB*, 8(13):2134–2145, 2015.

[47] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[48] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. SIGMOD, pages 677–689, 2015.

[49] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, MIT, 1979.

[50] A. Rosenberg. Improving query performance in data warehouses. *Business Intelligence Journal*, 11, Jan. 2006.

[51] J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, Mar. 1980.

[52] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.*, 7(12):1035–1046, Aug. 2014.

[53] R. Shoup and D. Pritchett. The ebay architecture. SD Forum, November 2006.

[54] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[55] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: The end of a column store myth. SIGMOD, pages 731–742, 2012.

[56] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory OLTP databases. In *DaMon*, 2013.

[57] M. Stonebraker. New sql: An alternative to nosql and old sql for new oltp apps. BLOG@CACM, June 2011.

[58] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[59] Tandem Database Group. NonStop SQL, a distributed, high-performance, high-availability implementation of sql. Technical report, Tandem, Apr. 1987.

[60] T. Team. In-memory data management for consumer transactions the timesten approach. SIGMOD '99, pages 528–529, 1999.

[61] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[62] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *HPTS*, 1997.

[63] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. Distributed systems, vol. ii: distributed data base systems. chapter R*: an overview of the architecture, pages 435–461. 1986.

[64] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[65] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.